L11: Entry 25 of 54 File: USPT Aug 13, 2002

DOCUMENT-IDENTIFIER: US 6434628 B1
TITLE: Common interface for handling exception interface name with additional prefix and suffix for handling exceptions in environment services patterns

Drawing Description Text (8):
FIG. 6 is a flow diagram depicting considerations to be taken into consideration when identifying the core technologies to be used in an architecture;

Drawing Description Text (42):
FIG. 40 is a diagram of an Application Model which illustrates how the different types of Partitioned Business Components might interact with each other;

Drawing Description Text (45):
FIG. 43 illustrates this Business Component Identifying Methodology including both Planning and Delivering stages;

Drawing Description Text (130):
FIG. 128 illustrates a flowchart for a method for structuring validation rules to be applied to a user interface for maximum maintainability and extensibility in accordance with an embodiment of the present invention;

Drawing Description Text (133):
FIG. 131 illustrates a validation rule class diagram;

Drawing Description Text (134):
FIG. 132 illustrates a rule validation interaction diagram;

Drawing Description Text (195):
FIG. 193 illustrates the Separate Models for Separate Business LUWs;

Detailed Description Text (16):
Frameworks also represent a change in the way programmers think about the interaction between the code they write and code written by others. In the early days of procedural programming, the programmer called libraries provided by the operating system to perform certain tasks, but basically the program executed down the page from start to finish, and the programmer was solely responsible for the flow of control. This was appropriate for printing out paychecks, calculating a mathematical table, or solving other problems with a program that executed in just one way.

Detailed Description Text (22):
There are three main differences between frameworks and class libraries: Behavior versus protocol. Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined, including rules for what a programmer is supposed to provide versus what the framework provides. Call versus override. With a class library, the code the programmer instantiates objects and calls their member functions. It's possible to instantiate and call objects in the same way with a framework (i.e., to treat the framework as a class library), but to take full advantage of a framework's reusable design, a programmer typically writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. Writing a program involves dividing responsibilities among the various pieces of software that are called by the framework rather than specifying how the different pieces should work together. Implementation versus design. With class libraries, programmers reuse only implementations, whereas with frameworks, they reuse design. A framework embodies the way a family of related programs or pieces of software

work. It represents a generic design solution that can be adapted to a variety of specific problems in a given domain. For example, a single framework can embody the way a user interface works, even though two different user interfaces created with the same framework might solve quite different interface problems.

Detailed Description Text (27):
Sun's Java language has emerged as an industry-recognized language for "programming the Internet." Sun defines Java as: "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language. Java supports programming for the Internet in the form of platform-independent Java applets." Java applets are small, specialized applications that comply with Sun's Java Application Programming Interface (API) allowing developers to add "interactive content" to Web documents (e.g., simple animations, page adornments, basic games, etc.). Applets execute within a Java-compatible browser (e.g., Netscape Navigator) by copying code from the server to client. From a language standpoint, Java's core feature set is based on C++. Sun's Java literature states that Java is basically, "C++ with extensions from Objective C for more dynamic method resolution."

Detailed Description Text (105):
During a high-level architecture design, help the user identify architecture services the user will need to address, by providing a logical level discussion one can use to assess types of base services and products needed for the specific situation.

Detailed Description Text (111):
This section should assist an architect in understanding the characteristics of, and the implications from selecting, a specific technology generation. The strengths and weaknesses of each technology generation should be understood when planning and designing a system. When identifying the core technologies to be used in an architecture, a view of the client's existing IT architecture 600, guiding principles 602 and business imperatives 604 should be taken into consideration, as depicted in FIG. 6.

Detailed Description Text (123):
The following sections identify the main characteristics associated with a Netcentric, Client Server or Host based technology generation. This list should in no way be considered complete and exhaustive but is included as a starting point from which the identification process may begin.

Detailed Description Text (126):
The following details the importance of each of the statements in FIG. 7 and should assist one in identifying the appropriate answer for the specific client engagement.

Detailed Description Text (133):
The following section details the importance of each of the statements found in FIG. 8 and should assist one in identifying the appropriate answer for your specific client engagement.

Detailed Description Text (136):
IT Guiding Principles 804 G1. The client maintains their applications internally and the IT department has the necessary resources, organizations and processes to maintain a Client Server application. Introduction of a Client Server application to a company's production environment can require a great deal of change to the Execution, Operations and Development architectures required to develop, run and support the production systems. Before a Client Server application is developed, it is important that the client identify how a system of this type will fit within the company's strategic technology plan.

Detailed Description Text (139):
The following section details the importance of each of the statements found in FIG. 9 and should assist you in identifying the appropriate answer for your specific client engagement.

Detailed Description Text (142):
IP Guiding Principles 904 G1. The Client has the resources, organizations and processes necessary for the development and operation of a Host based application. Before a Host based application is developed, it is important that the client identify how a system of this type will fit within the company's strategic technology plan. G2. Reliance upon a single vendor (IBM) for technology solutions is acceptable. Selection of a host based

architecture inherently locks the client into dependence upon one vendor for its technology solutions. While IBM is a reputable, stable company it may be important to ensure that the client's long term business strategy will be supported by IBM's technology vision and direction. G3. Centralized application and data is an acceptable strategy. A pure host based architecture eliminates the possibility of distributing data or business logic to the client. This removes some of the application performance benefits which can be seen by a distribution strategy, however, centralized access to the business logic and business data can improve operational stability and lower costs. A current trend is to transform mainframe based legacy systems into data- and application servers in a multi-tiered client/server or Netcentric architecture.

Detailed Description Text (161):
Netcentric Computing Top 10 Points Netcentric computing represents an evolution--it builds on and extends, rather than replaces, client/server. Netcentric computing has a greater impact on the entire business enterprise, hence greater opportunity and risk. Definitions of Netcentric may vary. One is about reach and content. Netcentric is not just electronic commerce; it can impact enterprises internally as well. You can begin identifying Netcentric opportunities for clients today. There are three basic types of Netcentric applications: advertise; inquiry; and fully interactive. One can underestimate the impact of Netcentric on infrastructure requirements. Build today's client/server engagements with flexibility to extend to Netcentric.

Detailed Description Text (173):
Two-tiered architecture describes a distributed application architecture in which business applications are split into front-ends (clients) and back-ends (servers). Such a model of computing began to surface in the late 1980s and is the prominent configuration in use today by companies which have attempted to migrate to client/server based computing.

Detailed Description Text (181):
A three-tiered architecture is often enhanced by the integration of distributed transaction processing middleware. This model of computing is often termed the "enhanced" client/server model. Most Netcentric architectures use a three- or four tiered approach with a web server and potentially a separate application server layer.

Detailed Description Text (184):
In contrast to mainframe and two-tiered client/server computing models, the principle advantage with a three-tiered enhanced client/server architecture is that it provides the benefits of a GUI application, but also provides a level of integrity and reliability found in mainframe centralized computing. That is, it will evolve to serve high-volume, high-integrity, and high-availability environments. Location and implementation transparency--The use of a transaction manager such as Tuxedo allows for service location independence. Distribution of logic to optimal resource--Since the application and database functions reside on their own physical devices, each can be optimally tuned for the work they perform. Database scaleable on throughput--In the enhanced three-tiered client/server model, client applications no longer connect directly to database servers. Instead, only application servers connect to the database servers. Security over service resources--With the application logic residing on back-end application servers, security over the applications is made possible at various levels. Redundancy and resiliency of services--A major disadvantage prominent in other models of computing is "single point of failure. Optimization of personnel resources--Developers can be utilized for specific talents in each tier. Allows for asynchronous and standardized messaging--The enhanced client/server model is really a superset of the RPC-based function shipping model which provides features such as asynchronous, event-driven programming. Administration, configuration, prioritization--The use of a transaction manager enables servers to be added, removed, or restarted dynamically. This allows for very robust, scaleable, and flexible applications.

Detailed Description Text (189):
Presentation Services enable an application to manage the human-computer interface. This includes capturing user actions and generating resulting events, presenting data to the user, and assisting in the management of the dialog flow of processing. FIG. 13 illustrates several components of the Presentation area of the Netcentric Architecture Framework.

Detailed Description Text (210):
In addition to the traditional tools (e.g., Visual C++, Visual Basic, PowerBuilder), Netcentric technologies have introduced new tools that can be used to develop Forms.

For example, a developer can use Symantec Visual Cafe to create a Java application that will execute directly on the users desktop without any interaction with a browser.

Detailed Description Text (247):
HTML's simplicity soon began to limit authors who demanded more advanced multimedia and page design capabilities. Enter Dynamic HTML DHTML. As an extension of HTML, DHTML allows Web pages to function more like interactive CD-ROMs by responding to user-generated events. DHTML allows Web page objects to be manipulated after they have been loaded into a browser. This enables users to shun plug-ins and Java applets and avoid bandwidth-consuming return trips to the server. For example, tables can expand or headers can scurry across the page based on a user's mouse movements.

Detailed Description Text (250):
HTML 4.0 and Dynamic HTML have given Web authors more control over the ways in which a Web page is displayed. But they have done little to address a growing problem in the developer community: how to access and manage data in Web documents so as to gain more control over document structure. To this end, leading Internet developers devised Extensible Markup Language (XML), a watered-down version of SGML that reduces its complexity while maintaining its flexibility. Like SGML, XML is a meta-language that allows authors to create their own customized tags to identify different types of data on their Web pages. In addition to improving document structure, these tags will make it possible to more effectively index and search for information in databases and on the Web.

Detailed Description Text (251):
XML documents consist of two parts. The first is the document itself, which contains XML tags for identifying data elements and resembles an HTML document. The second part is a DTD that defines the document structure by explaining what the tags mean and how they should be interpreted. In order to view XML documents, Web browsers and search engines will need special XML processors called "parsers." Currently, Microsoft's Internet Explorer 4.0 contains two XML parsers: a high-performance parser written in C++ and another one written in Java.

Detailed Description Text (295):
An image map menu can be useful where all users share some visual model for how business is conducted, and can be very engaging, but also painfully slow if even a moderate speed communications connection is required. Additional Image Map Services are required to map the location of user mouse clicks within the image to the corresponding page or window which is to be launched.

Detailed Description Text (393):
Storage Services manage the document physical storage. Most document management products store documents as objects that include two basic data types: attributes and content. Document attributes are key fields used to identify the document, such as author name, created date, etc. Document content refers to the actual unstructured information stored within the document. Generally, the documents are stored in a repository using one of the following methods: Proprietary database--documents (attributes and contents) are stored in a proprietary database (one that the vendor has specifically developed for use with their product). Industry standard database--documents (attributes and contents) are stored in an industry standard database such as Oracle or Sybase. Attributes are stored within traditional database data types (e.g., integer, character, etc.); contents are stored in the database's BLOB (Binary Large Objects) data type. Industry standard database and file system--Documents' attributes are stored in an industry standard database, and documents' contents are usually stored in the file-system of the host operating system. Most document management products use this document storage method, because today, this approach provides the most flexibility in terms of data distribution and also allows for greater scalability.

Detailed Description Text (434):
TelAlert; E-mail Systems e-mail systems--some e-mail systems and fax servers can be configured to generate pages to notify users when a defined event occurs such as e-mail/fax arriving. Telamon's TelAlert--TelAlert provides notification capabilities for UNIX systems. For example, it can page support personnel in the event of system problems.

Detailed Description Text (486):
Broadly defined, Messaging services enable information or commands to be sent between two or more recipients. Recipients may be computers, people, or processes within a

computer. Messaging Services are based on specific protocols. A protocol is a set of rules describing, in technical terms, how something should be done. Protocols facilitate transport of the message stream. For example, there is a protocol describing exactly what format should be used for sending specific types of mail messages. Most protocols typically sit "on top" of the following lower level protocol: TCP/IP--Transmission Control Protocol/Internet Protocol (TCP/IP) is the principle method for transmitting data over the Internet today. This protocol is responsible for ensuring that a series of data packets sent over a network arrive at the destination and are properly sequenced.

Detailed Description Text (500):
Function based middleware such as RPCs traditionally provide synchronous program control. Therefore, they tend to pass control from the client process to the server process. When this occurs, the client is dependent on the server and must wait to perform any additional processing until the servers response is received. This type of program control is also known as blocking. Some RPC vendors are enhancing their products to support asynchronous program control as well. What type of conversation control is required?

Detailed Description Text (560):
Microsoft Message Queue Server (MSMQ, formerly known as Falcon) Publish and Subscribe TibCo's Rendezvous TIB/Rendezvous' publish/subscribe technology is the foundation of TIBnet, TibCos solution for providing information delivery over intranets, extranets and the Internet. It is built upon The Information Bus.RTM. (TIB.RTM.) software, a highly scaleable messaging middleware technology based on an event-driven publish/subscribe model for information distribution. Developed and patented by TIBCO, the event-driven, publish/subscribe strategy allows content to be distributed on an event basis as it becomes available. Subscribers receive content according to topics of interest that are specified once by the subscriber, instead of repeated requests for updates. Using IP Multicast, TIBnet does not clog networks, but instead, provides for the most efficient real-time information delivery possible.

Detailed Description Text (563):
Streaming is an emerging technology. While some multimedia products use proprietary streaming mechanisms, other products incorporate standards. The following are examples of emerging standards for streaming protocols. Data streams are delivered using several protocols that are layered to assemble the necessary functionality. Real-time Streaming Protocol (RTSP)--RTSP is a draft Internet protocol for establishing and controlling on-demand delivery of real-time data. For example, clients can use RTSP to request specific media from a media server, to issue commands such as play, record and pause, and to control media delivery speed. Since RTSP simply controls media delivery, it is layered on top of other protocols, such as the following. Real-Time Transport Protocol (RTP)--Actual delivery of streaming data occurs through real-time protocols such as RTP. RTP provides end-to-end data delivery for applications transmitting real-time data over multicast or unicast network services. RTP conveys encoding, timing, and sequencing information to allow receivers to properly reconstruct the media stream. RTP is independent of the underlying transport service, but it is typically used with UDP. It may also be used with Multicast UDP, TCP/IP, or IP Multicast. Real-Time Control Protocol (RTCP)--RTP is augmented by the Real-Time Control Protocol. RTCP allows nodes to identify stream participants and communicate about the quality of data delivery.

Detailed Description Text (674):
Authentication can occur through various means: Basic Authentication--requires that the Web client supply a user name and password before servicing a request. Basic Authentication does not encrypt the password in any way, and thus the password travels in the clear over the network where it could be detected with a network sniffer program or device. Basic authentication is not secure enough for banking applications or anywhere where there may be a financial incentive for someone to steal someone's account information. Basic authentication is however the easiest mechanism to setup and administer and requires no special software at the Web client. ID/Password Encryption--offers a somewhat higher level of security by requiring that the user name and password be encrypted during transit. The user name and password are transmitted as a scrambled message as part of each request because there is no persistent connection open between the Web client and the Web server. Digital Certificates or Signatures--encrypted digital keys that are issued by a third party "trusted" organization (i.e. Verisign); used to verify user's authenticity. Hardware tokens--small physical devices that may generate a one-time password or that may be inserted into a card reader for authentication purposes. Virtual tokens--typically a file on a floppy or hard drive used for authentication (e.g. Lotus Notes ID file).

Biometric identification--the analysis of biological characteristics to verify individuals identify (e.g., fingerprints, voice recognition, retinal scans).

Detailed Description Text (717):
Encryption within the Transport Services layer is performed by encrypting the packets generated by higher level services (e.g., Message Transport) and encapsulating them in lower level packets (e.g., Packet Forwarding/Internetworking). (Note that encryption can also occur within the Communications Services layer or the Network Media layer.) Encryption within the Transport Services layer has the advantage of being independent of both the application and the transmission media, but it may make network monitoring and troubleshooting activities more difficult.

Detailed Description Text (742):
Media Access services manage the low-level transfer of data between network nodes. Media Access services perform the following functions: Physical Addressing--The Media Access service encapsulates packets with physical address information used by the data link protocol (e.g., Ethernet, Frame Relay). Packet Transfer--The Media Access service uses the data link communications protocol to frame packets and transfer them to another computer on the same network/subnetwork. Shared Access--The Media Access service provides a method for multiple network nodes to share access to a physical network. Shared Access schemes include the following: CSMA/CD--Carrier Sense Multiple Access with Collision Detection. A method by which multiple nodes can access a shared physical media by "listening" until no other transmissions are detected and then transmitting and checking to see if simultaneous transmission occurred. token passing--A method of managing access to a shared physical media by circulating a token (a special control message) among nodes to designate which node has the right to transmit. multiplexing--A method of sharing physical media among nodes by consolidating multiple, independent channels into a single circuit. The independent channels (assigned to nodes, applications, or voice calls) can be combined in the following ways: time division multiplexing (TDM)--use of a circuit is divided into a series of time slots, and each independent channel is assigned its own periodic slot. frequency division multiplexing (FDM)--each independent channel is assigned its own frequency range, allowing all channels to be carried simultaneously. Flow Control--The Media Access service manages the flow of data to account for differing data transfer rates between devices. For example, flow control would have to limit outbound traffic if a receiving machine or intermediate node operates at a slower data rate, possibly due to the use of different network technologies and topologies or due to excess network traffic at a node. Error Recovery--The Media Access service performs error recovery, which is the capability to detect and possibly resolve data corruption that occurs during transmission. Error recovery involves the use of checksums, parity bits, etc. Encryption--The Media Access service may perform encryption. (Note that encryption can also occur within the Communications Services layer or the Transport Services layer.) Within the Network Media Services layer, encryption occurs as part of the data link protocol (e.g. Ethernet, frame relay). In this case, all data is encrypted before it is placed on the wire. Such encryption tools are generally hardware products. Encryption at this level has the advantage of being transparent to higher level services. However, because it is dependent on the data link protocol, it has the disadvantage of requiring a different solution for each data link protocol.

Detailed Description Text (796):
Below are commonly used transaction monitors: BEA TUXEDO--provides a robust middleware engine for developing and deploying business-critical client/server applications. BEA TUXEDO handles not only distributed transaction processing, but also application and the full complement of services necessary to build and run enterprise-wide applications. It enables developers to create applications that span multiple hardware platforms, databases and operating systems. IBMs CICS/6000--an application server that provides industrial-strength, online transaction processing and transaction management for mission-critical applications on both IBM and non-IBM platforms. CICS manages and coordinates all the different resources needed by applications, such as RDBMSs, files and message queues to ensure completeness and integrity of data. Transarcs Encina--implements the fundamental services for executing distributed transactions and managing recoverable data, and various Encina extended services, which expand upon the functionality of the toolkit to provide a comprehensive environment for developing and deploying distributed transaction processing. Microsofts Transaction Server (Viper)--a component-based transaction processing system for developing, deploying, and managing high performance, and scalable enterprise, Internet, and intranet server applications. Transaction Server defines an application programming model for developing distributed, component-based applications. It also provides a run-time infrastructure for deploying and managing these applications.

Detailed Description Text (832):
Environment Verification Services ensure functionality by monitoring, identifying and validating environment integrity prior and during program execution. (e.g., free disk space, monitor resolution, correct version). These services are invoked when an application begins processing or when a component is called. Applications can use these services to verify that the correct versions of required Execution Architecture components and other application components are available.

Detailed Description Text (852):
Primarily there are three types of errors: system, architecture and application. System errors occur when the application is being executed and some kind of serious system-level incompatibility is encountered, such as memory/resource depletion, database access problems, network problems or printer related problems, because of which the application cannot proceed with its normal execution. Architecture errors are those which occur during the normal execution of the application and are generated in architecture functions that are built by a project architecture team to isolate the developers from complex coding, to streamline the development effort by re-using common services, etc. These architecture functions perform services such as database calls, state management, etc. Application errors are also those which occur during the normal execution of the application and are generally related to business logic errors such as invalid date, invalid price, etc.

Detailed Description Text (865):
Codes Table Services enable applications to utilize externally stored parameters and validation rules. For example, an application may be designed to retrieve the tax rate for the State of Illinois. When the user enters "Illinois" on the screen, the application first validates the user's entry by checking for its existence on the "State Tax Table", and then retrieves the tax rate for Illinois. Note that codes tables provide an additional degree of flexibility. If the tax rates changes, the data simply needs to be updated; no application logic needs to be modified.

Detailed Description Text (918):
Report Services are facilities for simplifying the construction and delivery of reports or generated correspondence. These services help to define reports and to electronically route reports to allow for online review, printing, and/or archiving. Report Services also support the merging of application data with pre-defined templates to create letters or other printed correspondence. Report Services include: Driver Services. These services provide the control structure and framework for the reporting system. Report Definition Services. These services receive and identify the report request, perform required validation routines, and format the outputted report(s). After the request is validated, the report build function is initiated. Report Build Services. These services are responsible for collecting, processing, formatting, and writing report information (for example, data, graphics, text). Report Distribution Services. These services are responsible for printing, or otherwise distributing, the reports to users.

Detailed Description Text (921):
The following types of reports are supported by the reporting application framework: Scheduled: Scheduled reports are generated based upon a time and/or date requirement. These reports typically contain statistical information and are generated periodically (invoices and bills, for example). On-demand: Some reports will be requested by users with specific parameters. The scheduling of these reports, the formatting, and/or the data requirements are not known before the request is made, so these factors must be handled at request time. Event-driven: This report type includes reports whose generation is triggered based on a business or system event. An example here would be a printed trade slip.

Detailed Description Text (925):
The report initiation function is the interface for reporting applications into the report architecture. The client initiates a report request to the report architecture by sending a message to the report initiation function. The responsibility of report initiation is to receive, identify, and validate the request and then trigger the report build process. The main components of reporting initiation are the following. Receive, identify, and validate a report request. The identification function determines general information about the request, such as report type, requester, quantity to be printed, and requested time. Based on the report type, a table of reports is examined in order to gather additional report-specific information and perform required validation routines for the report request. After the report

identification and validation functions have been successfully completed, the reporting process can continue. If any errors are identified, the report initiation function will return an error message to the requester application. Initiate report execution. The initiate report execution function processes the report profile and specific distribution requirements and determines the report to be created. It then passes control to the report execution process.

Detailed Description Text (942):
The requester ID, report name, and date/time are used to uniquely identify the report. These values are passed to APIs which request report status, print or delete a previously generated report.

Detailed Description Text (946):
Report processing is message-driven. Each defined API sends a unique message to the report process. The report process reads the messages from a queue and invokes the appropriate modules to handle each request. Subsequent process flows differ based upon the requested service. In the case of a report generation request, the process flow proceeds as follows: A record is added to the report status table. A message is sent to the report writer process for immediate generation or to the event manager for generation at a specified time (report scheduling). The appropriate application report writer module generates the report, prints it if specified in the original API request, and updates the status in the report status table.

Detailed Description Text (951):
FIG. 32 shows the module hierarchy for the custom report process. The Figure shows the relationships between modules, not their associated processing flows. It should be used to identify the calling module and the called modules for the process. FIG. 32 illustrates the Architecture Manager library 3200 which supports the report process.

Detailed Description Text (958):
Request Report. The Request Report function is responsible for processing report request messages written to the report process queue. It creates a new entry in the report status table with a status of "requested" and initiates the report writer process for immediate generation or sends a message to the event manager for future report generation.

Detailed Description Text (960):
Print Report. The Print Report function sends a generated report output file to a specified or default printer. The report name and requesting process ID is passed to identify the report.

Detailed Description Text (968):
Workflow services control and coordinate the tasks that must be completed in order to process a business event. For example, at XYZ Savings and Loan, in order to receive a promotion, you must complete an essay explaining why you should be promoted. This essay and your personnel file must be routed to numerous individuals who must review the material and approve your promotion. Workflow services coordinate the collection and routing of your essay and your personnel file.

Detailed Description Text (969):
Workflow enables tasks within a business process to be passed among the appropriate participants, in the correct sequence, and facilitates their completion within set times and budgets. Task definition includes the actions required as well as work folders containing forms, documents, images and transactions. It uses business process rules, routing information, role definitions and queues. Workflow functionality is crucial for the customer service and engineering applications to automate the business value chains, and monitor and control the sequence of work electronically.

Detailed Description Text (973):
Workflow can be further divided into the following components: Role management Role management ie provides for the assignment of tasks to roles which can then be mapped to individuals. A role defines responsibilities which are required in completing a business process. A business worker must be able to route documents and folders to a role, independent of the specific person, or process filling that role. For example, a request is routed to a supervisor role or to Purchasing, rather than to "Mary" or "Tom." If objects are routed to Mary and Mary leaves the company or is reassigned, a new recipient under a new condition would have to be added to an old event. Roles are also important when a number of different people have the authority to do the same work, such as claims adjusters; just assign the request to the next available person.

In addition, a process or agent can assume a role; it doesn't need to be a person. Role Management Services provide this additional level of directory indirection. Route management Route management enables the routing of tasks to the next role, which can be done in the following ways: Serial--the tasks are sequentially performed; Parallel--the work is divided among different players; Conditional--routing is based upon certain conditions; and Ad hoc--work which is not part of a predefined process. Workflow routing services route "work" to the appropriate workflow queues. When an application completes processing a task, it uses these services to route the work-in-progress to the next required task or tasks and, in some cases, notify interested parties of the resulting work queue changes. The automatic movement of information and control from one workflow step to another requires work profiles that describe the task relationships for completing various business processes. The concept of Integrated Performance Support can be exhibited by providing user access to these work profiles. Such access can be solely informational--to allow the user to understand the relationship between tasks, or identify which tasks need to be completed for a particular work flow--or navigational--to allow the user to move between tasks. Route Management Services also support the routing and delivery of necessary information (e.g., documents, data, forms, applications, etc.) to the next step in the work flow as needed. Rule Management A business process workflow is typically composed of many different roles and routes. Decisions must be made as to what to route to which role, and when. Rule Management Services support the routing of workflow activities by providing the intelligence necessary to determine which routes are appropriate given the state of a given process and knowledge of the organization's workflow processing rules. Rule Management Services are typically implemented through easily maintainable tables or rule bases which define the possible flows for a business event. Queue Management These services provide access to the workflow queues which are used to schedule work. In order to perform workload analysis or to create "to do lists" for users, an application may query these queues based on various criteria (a business event, status, assigned user, etc.). In addition, manipulation services are provided to allow queue entries to be modified. Workflow services allow users and management to monitor and access workflow queue information and to invoke applications directly. Is there a need for reporting and management facilities?

Detailed Description Text (977):
If rules or conditions can be identified which define the business process, with few exception conditions, workflow tools can then automate areas such as information routing, task processing, and work-in-process reporting. Are fixed delays or deadlines involved?

Detailed Description Text (983):
How an organization approaches the management of its workflow will determine which workflow management tools are appropriate to the organization. In general, there are three types of workflow, production, collaborative, and ad hoc. A production environment involves high transaction rates and thousands of documents in which the rules for a certain document can be defined for most of the time. Examples include accounts payable, insurance claims processing, and loan processing. A collaborative environment involves multiple departments viewing a single document with typically less number of documents than in the production environment. One example is a sales order. Ad hoc workflows arise from the specific temporary needs of a project team whose members become active and inactive depending on their function within the group. What is the relationship between the workflow and imaging components?

Detailed Description Text (991):
Business Logic is the core of any application, providing the expression of business rules and procedures (e.g., the steps and rules that govern how a sales order is fulfilled). As such, the Business Logic includes the control structure that specifies the flow for processing business events and user requests. There are many ways in which to organize Business Logic, including: rules-based, object-oriented, components, structured programming, etc. however each of these techniques include, although perhaps not by name, the concepts of: Interface, Application Logic, and Data Abstraction. FIG. 33 depicts the various components of the Business Logic portion of the Netcentric Architecture Framework.

Detailed Description Text (995):
Application Logic is the expression of business rules and procedures (e.g., the steps and rules that govern how a sales order is fulfilled). As such, the Application Logic includes the control structure that specifies the flow for processing for business events and user requests. The isolation of control logic facilitates change and adaptability of the application to changing business processing flows.

Detailed Description Text (999):
It is important to decide whether the business logic will be separate from the presentation logic and the database access logic. Today separation of business logic into its own tier is often done using an application server. In this type of an environment, although some business rules such as field validation might still be tightly coupled with the presentation logic, the majority of business logic is separate, usually residing on the server. It is also important to decide whether the business logic should be packaged as components in order to maximize software re-use and to streamline software distribution.

Detailed Description Text (1003):
Currently, Internet applications house the majority of the business processing logic on the server, supporting the thin-client model. However, as technology evolves, this balance is beginning to shift, allowing business logic code bundled into components to be either downloaded at runtime or permanently stored on the client machine. Today, client side business logic is supported through the use of Java applets, JavaBeans, Plug-ins and JavaScript from Sun/Netscape and ActiveX controls and VBScript from Microsoft.

Detailed Description Text (1005):
It is important to decide whether the business logic will be separate from the presentation logic and the database access logic. Today separation of business logic into its own tier is often done using an application server. In this type of an environment, although some business rules such as field validation might still be tightly coupled with the presentation logic, the majority of business logic is separate, usually residing on the server. It is also important to decide whether the business logic should be packaged as components in order to maximize software re-use and to streamline software distribution.

Detailed Description Text (1009):
Currently, Internet applications house the majority of the business processing logic on the server, supporting the thin-client model. However, as technology evolves, this balance is beginning to shift, allowing business logic code bundled into components to be either downloaded at runtime or permanently stored on the client machine. Today, client side business logic is supported through the use of Java applets, JavaBeans, Plug-ins and JavaScript from Sun/Netscape and ActiveX controls and VBScript from Microsoft.

Detailed Description Text (1014):
A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns. Patterns are a more formal way to document codified knowledge, or rules-of-thumb.

Detailed Description Text (1015):
Patterns represent the codified work and thinking of our object technology experts. While experts generally rely on mental recall or rules-of-thumb to apply informal patterns as opportunities are presented, the formalization of the patterns approach allows uniform documentation and transfer of expert knowledge.

Detailed Description Text (1017):
Patterns are usually concerned with some kind of architecture or organization of constituent parts to produce a greater whole. Richard Gabriel, author of Patterns of Software: Tales From the Software Community, provides a clear and concise definition of the term pattern: Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves. As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves. As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant. The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when one must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which may generate that thing.

Detailed Description Text (1021):
Component systems model--how the business works Component-orientation is a strategic

technology that may significantly impact a user's practice and clients. Component technologies are a natural evolution from object-oriented systems providing a more mature way of packaging reusable software units. Object-oriented systems more closely support business integration framework for solution delivery by shifting design focus away from an underlying technology toward a company's business conduct and functional behaviors. Business entities are represented as objects, which package data and functional behavior. This is in distinct contrast to traditional development approaches that maintain a ubiquitous split between functional behaviors and data.

Detailed Description Text (1027):
Component and object technology impacts most aspects of software development and management. Component technology is a new technology and a driving influence in the evolution of object-oriented (OO) methodologies. The Management Considerations section of the Introduction to Component-Based Development uses the Business Integration (BI) Model to discuss the impact of OO, including: Strategy and planning with a long-term view towards building reusable, enterprise software assets. Technology and architecture approaches for building cohesive, loosely coupled systems that provide long-term flexibility. Processes that shift analysis/design techniques from functional, procedural decomposition to business process modeling. These techniques are then used to decompose the system into domain objects and processes. People and organization strategies that emphasize greater specialization of skills within structures that support inter-team collaboration.

Detailed Description Text (1045):
Experience has shown that it's quite common for people to view components from different perspectives, as illustrated in FIG. 35. Some of them--typically designers--take a logical perspective. They view components as a means for modeling real-world concepts in the business domain. These are Business Components. Others--typically developers--take a physical perspective. They view components as independent pieces of software, or application building blocks, that implement those real-world business concepts. These are Partitioned Business Components. Developers also emphasize that Partitioned Business Components can be built from other independent pieces of software that provide functionality that is generally useful across a wide range of applications. These are Engineering Components.

Detailed Description Text (1048):
Business Components represent real-world concepts in the business domain. They encapsulate everything about those concepts including name, purpose, knowledge, behavior, and all other intelligence. Examples include: Customer, Product, Order, Inventory, Pricing, Credit Check, Billing, and Fraud Analysis. One might think of a Business Component as a depiction or portrait of a particular business concept, and as a whole, the Business Component Model is a depiction or portrait of the entire business. It's also important to note that although this begins the process of defining the application architecture for a set of desired business capabilities, the applicability of the Business Component Model extends beyond application building.

Detailed Description Text (1049):
Whereas Business Components model real-world concepts in the business domain, Partitioned Business Components implement those concepts in a particular environment. They are the physical building blocks used in the assembly of applications. As independent pieces of software, they encapsulate business data and operations, and they fulfill distinct business services through well-defined interfaces. Business Components are transformed into Partitioned Business Components based on the realities of the technical environment: distribution requirements, legacy integration, performance constraints, existing components, and more. For example, a project team might design an Order Business Component to represent customer demand for one or more products, but when it's time to implement this concept in a particular client/server environment, it may be necessary to partition the Order Business Component into the Order Entry component on the client and the Order Management component on the server. These are Partitioned Business Components.

Detailed Description Text (1051):
Components are useful throughout the development process. As a design artifact, early in the process, Business Components provide an underlying logical framework for ensuring flexibility, adaptability, maintainability, and reusability. They serve to break down large, complex problems into smaller, coherent elements. They also model the business in terms of the real-world concepts that make up the domain (e.g., entities, business processes, roles, etc.). Thus they provide the application with conceptual integrity. That is, the logical Business Components serve as the direct link between

the real-world business domain and the physical application. An important goal is to build an application that is closely aligned with the business domain. Later in the process, Partitioned Business Components and Engineering Components provide a means for implementing, packaging, and deploying the application. They also open the door to improved integration, interoperability, and scalability.

Detailed Description Text (1053):
In the Business Architecture stage 3604, a project team begins to define the application architecture for an organization's business capabilities using Business Components. Business Components model real-world concepts in the business domain (e.g., customers, products, orders, inventory, pricing, credit check, billing, and fraud analysis). This is not the same as data modeling because Business Components encapsulate both information and behavior. At this point in the process, an inventory of Business Components is sufficient, along with a definition, list of entities, and list of responsibilities for each Business Component.

Detailed Description Text (1054):
In Capability Analysis 3606 and the first part of Capability Release Design 3608, the project team designs Business Components in more detail, making sure they satisfy the application requirements. The team builds upon its previous work by providing a formal definition for each Business Component, including the services being offered. Another name for these services is "Business Component Interfaces." The team also models the interactions between Business Components.

Detailed Description Text (1055):
Throughout the remainder of Capability Release Design and into Capability Release Build and Test 3610, Business Components are transformed into Partitioned Business Components based on the realities of the technical environment. These constraints include distribution requirements, legacy integration, performance constraints, existing components, and more. Furthermore, to ensure the conceptual integrity of the Business Component model, a given Partitioned Business Component should descend from one and only one Business Component. In other words, it should never break the encapsulation already defined at the Business Component level. Also at this time, the project team designs the internal workings of each Partitioned Business Component. This could mean the Engineering Components that make up the Partitioned Business Component, the "wrapper" for a legacy or packaged system, and other code.

Detailed Description Text (1060):
This line of thinking leads to two types of Business Components: entity-centric and process-centric. Unfortunately, what commonly results from this paradigm is an argument over whether or not a particular Business Component is entity-centric or process-centric. In reality, Business Components are always a blend of both information and behavior, although one or the other tends to carry more influence. An appropriate mental model is a spectrum of Business Components.

Detailed Description Text (1061):
Business Components on the entity-centric side of the spectrum tend to represent significant entities in the business domain. Not only do they encapsulate information, but also the behaviors and rules that are associated with those entities. Examples include: Customer, Product, Order, and Inventory. A Customer Business Component would encapsulate everything an organization needs to know about its customers, including customer information (e.g., name, address, and telephone number), how to add new customers, a customer's buying habits (although this might belong in a Customer Account component), and rules for determining if a customer is preferred.

Detailed Description Text (1062):
Business Components on the process-centric side of the spectrum tend to represent significant business processes or some other kind of work that needs to be done. Not only do they encapsulate behaviors and rules, but also the information that is associated with those processes. Examples include: Pricing, Credit Check, Billing, and Fraud Analysis. A Pricing Business Component would encapsulate everything an organization needs to know about how to calculate the price of a product, including the product's base price (although this might belong in a Product component), discounts and rules for when they apply, and the calculation itself.

Detailed Description Text (1065):
A pattern emerges when one examines the way these Business Components interact with each other. Process-centric Business Components are "in control," while entity-centric Business Components do what they're told. To be more explicit, a process-centric

Business Component controls the flow of a business process by requesting services in a specific sequence according to specific business rules (i.e., conditional statements). The services being requested are generally offered by entity-centric Business Components, but not always. Sometimes process-centric Business Components trigger other process-centric Business Components.

Detailed Description Text (1066):
FIG. 37 shows how a Billing Business Component 3700 may create an invoice. The control logic 3702 (i.e., the sequence of steps and business rules) associated with the billing process is encapsulated within the Billing component itself. The Billing component requests services from several entity-centric Business Components, but it also triggers Fraud Analysis 3704, a process-centric Business Component, if a specific business rule is satisfied. Note also that "Step 6" is performed within the Billing component itself. Perhaps this is where the invoice is created, reflecting the design team's decision to encapsulate the invoice within the Billing component. This is one valid approach. Another is to model a separate entity-centric Invoice component that encapsulates the concept of invoice. This would effectively decouple the invoice from the billing process which might be a good thing depending on the requirements.

Detailed Description Text (1069):
As mentioned above, a User Interface Component is the implementation of a business process that is user controlled, but more explicitly it is a set of functionally related windows that supports the process(es) performed by one type of user. Examples include: Customer Service Desktop, Shipping Desktop, and Claim Desktop. These are not to be confused with low-level user interface controls (e.g., Active X controls), rather User Interface Components are usually built from low-level user interface controls. The reason for the dashed arrow in the diagram above is a subtle one. It points to the fact that earlier in the development process User Interface Components are generally not modeled as process-centric Business Components. Instead, they typically originate from the workflow, dialog flow, and/or user interface designs. See FIG. 39, which illustrates the flow of workflow, dialog flow, and/or user interface designs 3902, 3904, 3906 to a User Interface Component 3908. This makes complete sense given their direct tie to user controlled business processes.

Detailed Description Text (1070):
FIG. 40 is a diagram of the Eagle Application Model which illustrates how the different types of Partitioned Business Components might interact with each other. Business Entity Components 4002 and Business Process Components 4004 typically reside on a server, while User Interface Components 4006 typically reside on a client.

Detailed Description Text (1081):
Business Components model entities and processes at the enterprise level, and they evolve into Partitioned Business Components that are integrated into applications that operate over a network. Consequently, they serve as an excellent first step in the development of scalable, distributed enterprise applications that map closely to the business enterprise itself (i.e., the way it operates and the information that defines it).

Detailed Description Text (1082):
Business Components model the business, and thus they enable applications to more completely satisfy the business needs. They also provide a business-oriented view of the domain and consequently a good way to scope the solution space. This results in a good context for making process and application decisions. Finally, Business Components provide a common vocabulary for the project team. They educate the team in what's important to the business.

Detailed Description Text (1083):
When modeled correctly, entity-centric Business Components represent the most stable elements of the business, while process-centric Business Components represent the most volatile. Encapsulating and separating these elements contributes to the application's overall maintainability.

Detailed Description Text (1086):
Partitioned Business Components are application building blocks. As an application modeling tool, they depict how various elements of an application fit together. As an application building tool, they provide a means for systems delivery.

Detailed Description Text (1088):
Business Components model the business. It sounds straightforward, but even with

experience it's a challenge to identify the right components and to design them for flexibility and reuse. Flexibility and reuse are certainly more achievable with Business Components, but they are not inherent to Business Components. To accomplish these goals, as the previous examples suggest, one must understand what's happening within the enterprise and across the industry. One must work with business experts who understand the factors that will influence the current and future evolution of the business domain. This will improve one's ability to anticipate the range of possible change (i.e., to anticipate the future). The Business Component Model will be more flexible and reusable if it is challenged by scenarios that are likely to take place in the future.

Detailed Description Text (1096):
Business Components should encapsulate concepts that are significant to the business domain. Of course, this is subjective, and it certainly varies by business domain. In fact, business domain experts, with help from component modelers, are in the best position to make this judgment.

Detailed Description Text (1099):
It's important to strike a balance, and keep in mind that the ideal size depends on the domain. If there's a question in one's mind, it makes sense to lean toward smaller components. It's easier to combine them than to break them up. What's the best way to identify Business Components?

Detailed Description Text (1101):
The following steps describe one technique for identifying Business Components. FIG. 43 illustrates this Business Component Identifying Methodology 4300 including both Planning and Delivering stages 4302, 4304: 1. Start with entity-centric Business Components. For example, the customer is a significant entity in most business domains, therefore a Customer component may be included. A Customer Business Component would encapsulate everything an organization needs to know about its customers, including customer information (e.g., name, address, and telephone number), how to add new customers, a customer's buying habits (although this might belong in a Customer Account component), and rules for determining if a customer is preferred. Entities themselves can be physical or conceptual. For example, customers and products are physical--you can touch them. Orders, on the other hand, are conceptual. An order represents a specific customer's demand for a product. You cannot touch that demand. 2. Look for process-centric Business Components next. Generally speaking, a process-centric Business Component controls the flow of a business process. For example, in the utility industry, a Billing component would process customer, product, pricing, and usage information into a bill. Sometimes one will find an entity associated with the process-in this case, a bill or invoice--but another option is to model this entity as a separate, entity-centric Business Component, thus decoupling it from the process. What's the best way to identify the responsibilities of a business component?

Detailed Description Text (1110):
This is a common point of confusion. From a logical perspective, the term "component model" is frequently used to refer to a Business Component Model in the same way that "object model" is used to refer to a business object model.

Detailed Description Text (1112):
A grocery store chain is creating an enterprise-wide Business Component model. Currently the individual stores do not record specific customer information. Consequently, a model based on today's requirements would not retain customer information.

Detailed Description Text (1113):
However, they are looking into preferred customer cards. Furthermore, while analyzing the industry, the project team reads about a competitor with a pharmacy and video rental service. In both cases, customer information becomes critical. So the project team creates scenarios describing how they would use customer information to support these requirements. They create one Business Component Model that supports both today's and tomorrow's view of the customer.

Detailed Description Text (1114):
In the near future, when the chain adopts preferred customer cards, and in the more distant future, ifthey decide to add a pharmacy or video rental service, the Business Component design for their current application will provide a solid foundation for the future requirement of tracking customer information. If they weren't using Business Components, they would not have a model that maps to their business domain, and

introducing new requirements would require more abrupt changes. Example: Inventory Management

Detailed Description Text (1147):
The close tie that component and object modeling enables between the software solution and business process may help software analysts and users or business analysts to better understand each other, reducing errors in communications. This represents a significant opportunity, because misunderstanding user requirements has been proven to be the most costly type of mistake in systems development. A component model further improves the understanding of the software design by providing a larger-grained model that is easier to digest.

Detailed Description Text (1161):
Component-based Systems are Distinguished by a Business Component Model The presence of a reusable business component model is a key characteristic

Detailed Description Text (1162):
A component-based software architecture may have a domain component model shared by the application processes. The component model contains the core business components that represent the business directly in software. These components perform behaviors upon request by windows, reports, or batch process control objects.

Detailed Description Text (1163):
The presence of a component model distinguishes component-based systems from procedural, client/server systems. In a procedural approach, there is no shared business component model. This typically requires, for example, programs to pass data to each other in a context record. Thus, any changes to the data may affect many programs. The extent of business logic reuse is also usually less with the procedural approach.

Detailed Description Text (1164):
The presence of a business component model also distinguishes a component-based architecture from that produced by componentware tools. Specifically, many traditional and even component-based tools provide data-aware controls that tie the user interface directly to the database. This is indeed a powerful technique to rapidly build simpler, less strategic applications. However, it suffers from a lack of smaller-grained business reuse and increased coupling between presentation and data. This may increase maintenance costs and miss opportunities to flexibly model complex business processes, as can be done with a component model. On the other hand, producing a reusable component model requires a higher level of abstraction and is therefore a more difficult approach. Component systems are based on standards

Detailed Description Text (1176):
Most traditional engagements divide roles into two basic categories, functional and technical, or architecture. Component-based development introduces a third dimension by requiring an extensive modeling role. Early experience has shown that the capability to draw abstractions in modeling a business problem or application framework is a unique skill set distinct from purely technical or functional skills. Managing the domain component model requires new organization approaches

Detailed Description Text (1177):
In addition, the extensive reuse of a core business component model requires an organization structure that manages it as a shared resource. This creates a tension between the needs to support consistent reuse of core components, and the desire to solve a business problem front-to-back. Experience has shown this often requires some form of matrix organization, combining vertical-based leadership along the lines of business functions, and horizontal-based leadership along the lines of architecture layers.

Detailed Description Text (1215):
It is important to note that component technology skills cover a wide range of competencies--from modeling and design skills to detailed programming syntax. Rarely may one individual have the necessary expertise in all these areas. Thus, experience has shown that it is necessary to find individuals that specialize in one of these areas to leverage across a large team. The key is obtaining the right balance of technology and methodology skills. One engagement used a 1:1:1 rule to leverage expertise

Detailed Description Text (1216):

One large engagement found the most effective leveraging ratio was 1:1:1, comprising an experienced object specialist, an experienced programmer without object skills, and an inexperienced person. Note that this 1/3 ratio rule only applied to the team doing implementation. Thus, even though the total team size was about 200, only 40-50 were doing hands-on implementation, implying the need for about 13-17 skilled people.

Detailed Description Text (1224):
However, software development managers must recognize that component technology has a pervasive impact on many aspects of the development process including estimating, planning, methodology, and technology architecture. For example, iteration impacts many of the standard rules-of-thumb for work completion. And the extensive reuse of a common business component model requires more sophisticated organization strategies. Managers must invest time in training

Detailed Description Text (1230):
Developing with component technology demands more consistency, because an application framework and business or domain component model provide more reuse. In particular, much of the business logic may be shared by a common domain component model, viewed by many windows. To strive for this greater level of reuse across many business functions requires coordination among many developers. The risk is that the components may not fit together.

Detailed Description Text (1234):
Because of the significant technical challenges often faced, a team may be tempted to staff all the experienced component developers on an architecture frameworks team. This strategy makes some sense. However, it should not be followed to the exclusion of leveraging the application or component modeling development team. Developing the functional business logic requires component development and methodology skills, as well.

Detailed Description Text (1259):
A component-based development project requires creativity. The overall atmosphere is usually very challenging with fewer, concrete rules. The answer to many analysis and design decisions is, "it depends". Similarly, the development environments encourage exploration and browsing.

Detailed Description Text (1267):
On-going support is necessary to help developers continue building skills. On-going training is important because the entire development lifecycle is affected, to some degree, by the shift to components. An individual's first few assignments should be carefully planned to enable growing skills, and to identify people who demonstrate aptitude. Time must also be allowed for scaling the productivity learning curve, after initial skills are developed. This generally requires a fair degree of commitment from experienced frameworks developers to provide mentoring.

Detailed Description Text (1270):
A skills certification process helped to: More rigorously identify and describe competencies of what is really desired in terms of skills and competence; and, what habits should be discouraged and flagged as performance problems. Track peoples' growth-it encourages improvement by challenging people. Provide a more effective way to assign appropriate roles to people and offer up the opportunity for people to grow into a more challenging role as quickly as they are adequately prepared. Support more effective communications of what resources had which skills (e.g., through a wallchart)

Detailed Description Text (1292):
Architecture roles must be defined to support this greater degree of specialization. One engagement used the following partitioning strategy: Functional architect-responsible for resolving decisions for what the system should do. This person is ideally a user with a solid understanding of systems, or a systems person with a good understanding of, and relationship with, the users. Technology architect-responsible for delivering the platform, systems software, and middleware infrastructure to support execution, development, and operations architectures. User interface architect-responsible for setting direction of the user interface metaphor, layout standards, and integrated performance support (IPS). Application frameworks architect-responsible for designing, delivering, and supporting the application framework that provides the overall structure, or template, of the application. Object model architect-responsible for identifying and resolving modeling issues necessary to achieve a high degree of business reuse and modeling consistency.

Detailed Description Text (1294):
One must be very careful in ensuring that application frameworks are not "over-architected". Experience has shown that many frameworks fall by the way-side for the simple reason that they were not built closely enough in conjunction with the application development. They become too theoretical, complicated and over-engineered making them performance bottlenecks and obstacles to simplifying the application architecture. It has been found that frameworks should "fall out" of the application domain as candidates become obvious. Experienced developers that work closely with the application can quickly identify repetitive constructs and abstract useful frameworks from this context. Data and object model architects must clearly define their roles

Detailed Description Text (1302):
FIG. 45 illustrates a traditional organization structure including an activities component 4502, a credit/collections component 4504, a billing component 4506, and a finance component 4510. This traditional organization for most projects is vertically organized based upon business function with a technology architecture team. In this organization model, there would be one or more developers that are responsible for building a business function end to end. This works well for the following reasons: aligns well with the business process and software decomposition enables clear work direction--i.e., complete a window or report, front-to-back ensures that complete functions work in an integrated, end-to-end fashion teams better align to application releases

Detailed Description Text (1307):
FIG. 47 illustrates a workcell organization approach including an activities component 4702, a credit/collections component 4704, a billing component 4706, and a finance component 4710. This approach combines the two previous approaches into a workcell. The primary orientation can be aligned either way, but a functional orientation seems more natural for a business application. A cell is comprised of a complete set of specialized skilts such as functional analyst, object modeler, application architect, and even user. Cross-cell architects then provide specialized direction for a particular role.

Detailed Description Text (1318):
Workcell alignment may be influenced by the needs of the client. If the clients objective is to develop a highly reusable business component model, then it may be appropriate to have a single team focused on developing the component model. On the hand, if the client is most concerned about delivering business functionality, workcells should be aligned by business function.

Detailed Description Text (1357):
Even when incremental development does not prove feasible for entire application releases, the approach can be effective on a smaller scale. For example, the development and release of a single application may require extensive integration of diverse behaviors in a reusable domain component model. The domain components must be put in place early to allow reuse; then, behaviors are incrementally added as the business use cases are analyzed and designed. As in the previous case, iteration naturally occurs; but, again, incremental proves to be a more acceptable metaphor.

Detailed Description Text (1361):
This model incorporates the idea of simultaneous top-down and bottom-up development. Much development effort may follow a relatively top-down, sequential approach. This includes analyzing and designing: the business environment and processes, domain model, and then application. Concurrently, an architecture effort proceeds bottom-up. This builds: the technology architecture of platform system software, hardware and infrastructure services; and then application architecture, or frameworks. Top-down and bottom-up efforts then conceptually meet in the middle, integrating the application framework with the application. Both the architecture and component model lead application development

Detailed Description Text (1363):
Starting the component model early is essential to enabling reuse of a consistent, cross-functional set of business components. These core domain components must be defined early, at least in preliminary form. Otherwise, the simultaneous integration of functionality from many windows or reports would be extremely chaotic. In addition, developers may implement business logic in the user interface layer, rather than in the business components where it can be reused. Furthermore, early design of the component model before user interface logic improves the odds of creating a pure component model,

decoupled from the interface.

Detailed Description Text (1377):
Different roles for team members require different development methodologies. For example, possible roles are: Application developer--responsible for implementing a particular business function, such as accepting bill payment. This focuses on the application-specific design and implementation tasks such as: working with a user to define requirements or use cases, designing the user interface, and implementing application functionality. Component Model developer--builds, refines, and supports the core, reusable business components in the application. Frameworks developer--responsible for the application and technology architecture that provide common services and control logic for the application.

Detailed Description Text (1379):
At the micro-level components make it more reasonable to execute more development tasks in parallel. Components enable this by providing more discrete work objects that are more clearly separated by their interfaces. Because interfaces are the contracts through which components interact, the internals of a component can be developed independently as long as the interfaces are respected. Dependencies on shared components need to be managed

Detailed Description Text (1383):
During implementation, detailed design and coding steps may overlap. However, the rules and guidelines for sequencing these should be spelled out in rigorous detail. Note that this does not imply iteration per se, although that may be a desirable side-effect if controlled. Rather, this approach merely suggests tactically interspersing the design and code activities, particularly to aid in--experienced developers in transitioning from design to code. Define concrete milestones with short intervals

Detailed Description Text (1386):
A previous point emphasized starting the component model development early, because many of these components are reused in many business functions. Thus, their preliminary structure must be available before multiple windows require their use. This implies that many different behaviors may need to be continuously integrated into these components over and over. The component model development, then, is very much event-driven like a factory. Incremental is a good term for continuous integration of behaviors in the component model

Detailed Description Text (1396):
Performance prototypes primarily address technology architecture questions. For example, the architecture team may need to decide early on whether to use messaging, remote procedure calls, or shipped SQL statements for distribution services between client and server. A prototype is often the only way to identify the most effective solution. Proof-of-concept prototypes address complexity

Detailed Description Text (1411):
Managing iteration is difficult but possible. Usually the plan must incorporate a hybrid of waterfall, incremental, and iterative models as appropriate. The right process depends on the organization or teams' skills, the degree of technical risk, and the specific application and business requirements.

Detailed Description Text (1437):
When developing components using objects, regression testing becomes even more important. For example, inheritance often results in sub-classes coupled to their parent. A parent class may have side effects with subtle implications to children, which are difficult to identify for test cases. Experience has shown that even seemingly innocuous changes to a parent can damage previously tested sub-classes.

Detailed Description Text (1466):
Each project using component-based technology determines how to use OO CASE tools to support an object-oriented methodology and its deliverables. These deliverables range from high-level business process documentation in the business-modeling phase to descriptions of classes in the construction phase. UML compliant CASE tools provide a number of the deliverables that most object methodologies uses, however, there are almost always some deliverables that do not fit in the selected tool. There is also a discrepancy with the CASE tools' ability to comply with UML due to the continuing evolution of UML versions.

Detailed Description Text (1491):

An object-oriented system must assign component ownership at multiple levels. Business process owners are still necessary; however, clear lines of responsibility must be assigned for the domain object model. Often these two may have a tight relationship. For example, consider a gas utility customer system that provides customer service orders. The service order business process and service order domain object owner should probably be the same person. However, the service order process may also need to collaborate with other key domain components such as the customer and premise. This requires collaborating and communicating with other developers. Rigid ownership boundaries may not work

Detailed Description Text (1492):
Experience has shown, however, that the level of communications with core business objects such as customer and bill account is so high that the rigid ownership might be ineffective. The resulting communications of requirements may produce inefficient hand-offs and bottlenecks. For large, mission-critical applications, multiple levels of ownership must then be defined. However, this creates a risk of conflicts. Before components mature, the rules of divisions should probably be more rigid. Later, multiple developers can modify common classes, while keeping responsibility to release, or publish, the code in the hands of a single owner.

Detailed Description Text (1493):
Thus, ownership roles may overlap, requiring the rules of engagement to be defined. Yet, every scenario cannot be spelled out precisely. The team and leadership must then be very participatory and flexible to adapt to the dynamic requirements. One large engagement defined separate, overlapping ownership responsibility for: Windows Domain object model sub-systems, or components; the model comprised about 350 model objects which were partitioned into about 12 major areas Business processes that were particularly complex, highly reusable, and cut across many windows; for example, writing off a bill Common architecture framework components Separate concept of ownership from developer for increasing productivity

Detailed Description Text (1524):
Tools are necessary to identify categories of similar behavior such as the class hierarchy, where used, senders of, implementors of, etc. Today, many environments for C++, Smalltalk, Visual Basic & Java provide robust browsers with this comprehensive functionality. Additionally case tools also provide search capabilities. Unfortunately every tool uses a different method for finding artifacts, such as text searches for documents, menu provided searches in case tools, and where used and senders of within browsers.

Detailed Description Text (1539):
This desktop tool integration strategy needs to take into account the comprehensive approach used by the configuration management strategies. In other words, relevant documents need to be associated with the components and business processes they update so that key stakeholders can subscribe to alarms that may make them aware of when sections of documentation need updating. This process may help ensure that the publishing model is dynamic and current.

Detailed Description Text (1547):
The timing of when to address performance may initially appear trivial. "Design performance in from the start" is one often-repeated rule. The opposing viewpoint is expressed by computer scientist David Knuth who said, "Premature performance tuning is the root of all evil". Timing when to address performance is actually a complicated management issue. The competing forces and their possible resolution are discussed further below.

Detailed Description Text (1593):
Opportunities for performance tuning are found both in bottlenecks and in distributed inefficiencies. There are generally many tools available in detecting bottlenecks. Distributed inefficiencies are usually more difficult to identify with tools. Whether performance optimizations are realized through cognitive analysis, or tool-assisted profiling, it is important to measure the gains against a baseline performance level.

Detailed Description Text (1609):
Because batch processing executes on a server and requires limited user interaction, many of the services used for on-line architectures are not needed. For example, the services used for distributing components--naming, distributed events, bridging, trader, etc--are not needed for a batch architecture. In fact, the interfaces that encapsulate components and provide location transparency can add significant overhead

to a batch architecture. To avoid the expense of unneeded services, the component stubs can be wrapped with a layer of indirection that short-circuits the normal distribution mechanisms. This will provide performance that will approximate local function calls.

Detailed Description Text (1610):
Typically, business objects have to be instantiated from a relational database (RDBMS) before the batch application can make use of them. This extra overhead is a very real concern. It is an unfortunate fact that in many ways the more "object-oriented" your design is, the worse it fits into the relational paradigm of rows and tables. For one thing, these designs tend to have lots of objects with embedded instances or references to other objects. And the primary reason that such designs have RDBMS performance problems is that in the database, resolving such an object relationship requires joins or recursive queries. When mapping from your object model to the RDBMS, there is a tendency to "normalize" your object over many tables, and the performance can easily plummet.

Detailed Description Text (1612):
For some applications, an LRU caching policy might not be the right choice; a more complicated scheme with multiple cache levels might be necessary. For this reason it would be best to make the caching policy itself be an object (consider the Strategy pattern for making an object from an algorithm) so you can change the policy on demand. Cache operations and accesses. One of the reasons component-based batch performs so poorly is due to the fact that, in order to maximize modularity and preserve encapsulation, a lot of operations are performed redundantly. For instance if a balance is implemented as a calculation, and if it is needed by six different objects it is recomputed six times. These situations are very easy to identify with a performance monitor that tells you where the program spends most of its time; it is not uncommon to find that most of the time is actually spent in very few methods. For these methods (and only for them!) cache the result in an instance variable. Every time the method is invoked, check if the instance variable contains an answer: if not, compute it and store it there; if yes, just return it. Of course, each operation on the object that invalidates the result of the computation must invalidate the cache too! This technique has a very small impact on your object design and typically leaves the interface unchanged. Cache objects. Typically, this would involve leaving recently referenced objects instantiated in memory for some length of time after their last use. Then, if the object is accessed again, you check the memory-resident cache before re-loading the object from the DBMS. Usually you would construct this cache as a hash table keyed by object ID, and use a LRU policy to keep the cache size manageable. Expect degraded performance if you do anything to destroy the utility of the cache. For some applications, LRU might not be the right choice; a more complicated scheme with multiple cache levels might be necessary. For this reason it would be best to make the caching policy itself be an object (consider the Strategy pattern for making an object from an algorithm) so you can change the policy on demand. Make use of "lazy" or "deferred" loading. That is, don't do a "deep" instantiation until you know you're going to use the associated parts of the object. Instead, load selected sub-objects only when first referenced This can save on memory overhead as well as DBMS access. In some cases you can use a hybrid strategy: do a "shallow" instantiation by default, but provide the client program with a way to build the complete object on demand to provide more deterministic performance. One thing to be careful of with this approach is that if you really do tend to use most parts of the object during high-volume processing, loading it in piecemeal can actually worsen the performance, because of the overhead of maintaining the load state and because of the smaller DBMS transactions sizes. These techniques have a very small impact on your object model. De-normalize your database where possible. Typically when one does object-to-relational mappings, one tends to make every unique object type a separate table. This is best from a design perspective. But in cases where you know you have a fixed set of "private" associations (meaning physical aggregation with no possibility of shared references), then fold that sub-object data into the enclosing object's RDBMS table. It's not pretty, but it can save lots of extra loading time. Also, look at ways to do aggregate loads based on some unique object ID. For example, if you have collection-valued sub-components, insert the object ID of the enclosing object in the sub-object tables and do aggregate loads in code, rather than doing a "point-of-use" instantiation for each one separately. Of course, these optimizations can have a more substantial impact on your object model. Consider making "light" versions of some of your objects. That is, for performance critical situations, create alternate implementations of your business objects that don't have all the baggage of the first-class objects. Yes, this can be ugly and more difficult to maintain. But for many batch processing applications you might find that you can drop a lot of the (persistence-related) complexity of an object without affecting the batch processing at all. Then create fast hand-tuned routines to

instantiate the "light" objects from the database.

Detailed Description Text (1634):
Most business systems today include some sort of batch processing. Batch processing is the execution of a series of instructions that do not require any interaction with a user to complete. Batch jobs are usually stored up during the day and executed during evening hours when the system load is typically lower.

Detailed Description Text (1668):
As a result of filter processing's incremental nature, one or more filters, tied together with pipes, define a process's Logical Unit of Work (LUW); i.e., the filters defining the steps of the process are sandwiched by the beginning and ending of the transaction. Expanding this model, each subsystem representing the LUW can be modeled as a filter with input and output that encompasses the internal filters. These filters are then tied together through the use of pipes to represent the system. In this manner, the Processing Pipeline pattern offers a consistent way to view the system that scales to whatever size and degree of complexity the system grows.

Detailed Description Text (1729):
If constants are obtained by other means than explicit language constructs like "public final int HOME_ADDRESS" than public accessors are used to insulate a client from changes in how the constant is obtained. In this case the values of each of the constants should be defined privately inside the Constant Class. Public accessors are then provided for clients to obtain the constant values. This allows for "changing constants". Business-related values that may seem constant at design and construction time very often are not. Some of these "constants" may eventually require some logic to determine their value. If clients obtain constants through accessor methods, no changes (except within the accessor) will have to be made if the logic is added. This is a particularly safe practice when programming rules dictate all constants to be stored and retrieved from database tables.

Detailed Description Text (1847):
From the example of FIG. 83, the following steps are shown: 1. The Client component wants to invoke some functionality, which is located on the legacy system. The Client sends a message via the Component Integration Architecture (e.g. ORB) on the way to the Legacy Wrapper Component. 2. The Component Integration Architecture (e.g. ORB) forwards the call to the appropriate Legacy Wrapper Component. 3. The Legacy Wrapper Component sends the call via the Component Adapter to the Legacy Integration Architecture. When necessary, the Component Adapter reformats the call parameters into an acceptable format for the Legacy System. 4. The Legacy Integration Architecture receives a call for the host-based Legacy application and forwards it to the Legacy Adapter. 5. The Legacy Adapter receives the message from the Legacy Integration Architecture and formats it to match the API of the Legacy System. It makes the appropriate calls on the Legacy System. The Legacy System executes the function and returns the results to the Legacy Adapter 6. The Legacy Adapter receives the results and returns them to the Legacy Integration Architecture. 7. The Legacy Integration Architecture receives the result and forwards it to the Legacy Wrapper Server Component through the Component Adapter. 8. The Legacy Wrapper Component receives the result, reformats the parameters for the component system and forwards it to the Component Integration Architecture. 9. Finally, Component Interaction Architecture receives the result and forwards it to the Client

Detailed Description Text (1941):
FIG. 100 illustrates a flowchart for a method 10000 for interfacing a naming service and a client with the naming service allowing access to a plurality of different sets of services from a plurality of globally addressable interfaces. In operation 10002, the naming service calls for receiving locations of the global addressable interfaces. As a result of the calls, proxies are generated based on the received locations of the global addressable interfaces in operation 10004. The proxies are received in an allocation queue where the proxies are then allocated in a proxy pool (see operations 10006 and 10008). Access to the proxies in the proxy pool is allowed for identifying the location of one of the global addressable interfaces in response to a request received from the client in operation 100010.

Detailed Description Text (1987):
Additionally, each system must implement a message language. The message language defines the rules for writing and interpreting the descriptive meta-data. It describes how the meta-data is parametarised and embedded in the message.

Detailed Description Text (2084):
Presentation Services enable an application to manage the human-computer interface. This includes capturing user actions and generating resulting events, presenting data to the user, and assisting in the management of the dialog flow of processing. The Presentation Services forward on the user requests to business logic on some server. Typically, Presentation Services are only required by client workstations.

Detailed Description Text (2091):
A plurality of presentation interfaces may be interfaced. Optionally, a model may be interfaced for management purposes. With such an option, the model may further include a proxy. As another option, errors and exceptions may also be handled. As a third option, events intended to be received may be triggered by the presentation interface.

Detailed Description Text (2095):
To complete a single use case, the client may need to interact with a number of server components. From the user's perspective, one unit of work is being performed but it may involve multiple, discrete interfaces and multiple server invocations. Some business logic is required to manage the complex flow to complete this unit of work. For example, suppose a use case for a network inventory management system is "Add Network Card". This may require the user to input information in three or four screens and client communication with more than one server component. Managing this flow is not the responsibility of the presentation logic but still needs to be executed on the client.

Detailed Description Text (2100):
An Activity is responsible for: managing client logical units of work maintaining client representation of a business model validation across multiple interfaces (complex business logic) error and exception handling communication with server and other services creating other Activities triggering events intended to be "caught" and acted on by the presentation logic

Detailed Description Text (2101):
An Activity resides between the actual user interface and the business model and server components as shown in the Entity Relationship diagram below:

Detailed Description Text (2118):
FIG. 128 illustrates a flowchart for a method 12800 for structuring validation rules to be applied to a user interface for maximum maintainability and extensibility. In operations 12802 and 12804, a plurality of user interface widgets are provided along with a plurality of validation rules which govern use of the user interface widgets. A user is allowed in operation 12806 to select the validation rules to associate with the user interface widgets of a first user interface. The validation rules of the user interface widgets of the first user interface are automatically associated across a plurality of separate different user interfaces in operation 12808.

Detailed Description Text (2119):
The validation rules may be created at the time the first user interface is created. As another option, the validation rules may be implemented by a different class for each type of validation. As a further option, an indicator may be displayed upon one of the validation rules being violated. Additionally, each validation rule class may extend an abstract validation rule class that defines what types of widgets are supported. Also, a request for the validation rules may optionally be received from one of the user interfaces.

Detailed Description Text (2120):
How can you structure validation rules to be applied to a user interface for maximum maintainability and extensibility.

Detailed Description Text (2121):
Imagine a typical Windows or web-based client/server application. In most cases where a "windows" type of user interface is provided, an application supports some business rules by validating data entered by the user. A common example of this is checking the format of data in an entry field or ensuring that a required field is not left empty.

Detailed Description Text (2122):
The business rules supported by user interface validation is usually somewhat limited. The scope of these rules is generally constrained to checking if a field is empty, checking the format of a field (date, time, numeric, currency, etc.), and checking if a field has alpha-characters, numeric-characters, or both. In addition, due to fact that many widgets provide constraints through their own form (list boxes, radio buttons),

the types of widgets that require this type of validation checking is also somewhat limited (text fields, editable combo boxes, etc.).

Detailed Description Text (2124):
Because this type of validation will most likely be required across all of an application's user interfaces and the fact that the types of validation rules and widgets needed to validate are limited, this behavior is a strong candidate for a framework.

Detailed Description Text (2125):
The framework would provide a common approach to validating user data across all of an application's user interfaces. Rules would be applied consistently throughout the application. While some common validation rules would be provided, the framework needs to allow their behavior to be modified (overridden) and make it easy for new rules to be added.

Detailed Description Text (2127):
Therefore, for each user interface in an application, encapsulate their validation logic in a User Interface Validator. FIG. 130 illustrates a user interface validator association diagram. A User Interface Validator 13000 associates various validation rules with the user interface widget they are to be applied to.

Detailed Description Text (2129):
The rules are implemented by a different class for each type of validation. Each of these validation rule classes must know how to check its rule for every type of widget that can be checked. As mentioned in the Context section of this pattern, this will most likely be limited to text entry type widgets. In addition, each validation rule class extends an abstract validation rule class that defines what types of widgets are supported. This is an implementation of the Visitor pattern.

Detailed Description Text (2130):
FIG. 131 illustrates a validation rule class diagram.

Detailed Description Text (2131):
Note that the check operations accept a Validate 13200 type of class. Each widget that can be validated with this framework must implement a validateRule method. This simple method accepts some ValidationRule 13202 as a parameter and simply turns around and calls the check method on the rule passing itself in as a parameter. This interaction is shown in FIG. 132, which illustrates a rule validation interaction diagram.

Detailed Description Text (2132):
The concrete implementation of the check method will be invoked. This method knows how to extract the data from the particular widget provided and verify the rule.

Detailed Description Text (2133):
The User Interface Validator's job is to associate these rule instances with all of the widgets they pertain to. When the validate method is invoked on the Validator, all of the rules are sent to each of the appropriate widgets via the validateRule method.

Detailed Description Text (2134):
New rules can be added by creating new classes that extend off of the abstract Validation Rule class. No changes need to be made to the widgets.

Detailed Description Text (2135):
Benefits Consistency. All user interface validation rule checking is done in the same way using the same rule logic. Extensibility. New rules can be added without affecting any other part of the application. Automation. Application of validation rules can be automated with a GUI based tool rather easily.

Detailed Description Text (2136):
The associations between a widget and the rule to apply to it should be set up when the user interface is created. A user interface can implement a method that accepts a rule and widget and passes it on to the User Interface Validator as shown in the code example below: ValidateTextField userNameField=new TextField("user name"); ValidateTextField passwordField=new TextField("password"); ValidateTextArea commentsArea=new TextArea("comments); this.addValidation(userNameField, new NotEmptyValidationRule( )); this.addValidation(passwordField, new NotEmptyValidationRule( )); this.addValidation(passwordField, new NotNumericValidationRule( )); this.addValidation(commentsArea, new

MaxLengthValidationRule(255));

Detailed Description Text (2139):
In the above code, three widgets are created and then associated with various validation rules. The user name and password fields are required and cannot be left blank, the password may not contain any numbers, and the comments text area may not be longer than 255 characters.

Detailed Description Text (2140):
Note that each of the widgets is created with a string that describes a name for the widget that the user would recognize. This name is used in the error list to help a user identify which widget failed validation.

Detailed Description Text (2141):
At some appropriate time, the user interface sends the validate message to the User Interface Validator. This method steps through each of the rules provided to it when the user interface initialized and passes them to their associated widget by the validateRule method. The code is shown below:

Detailed Description Text (2144):
Visitor Each of the rules are implemented as a Visitor according to the GoF pattern of the same name.

Detailed Description Text (2146):
FIG. 133 illustrates a flowchart for a method 13300 for assigning a view to an activity. Notification is received that a startup event of an activity has occurred in operation 13302. A reference to a first instance of an object created by the startup event of the activity is also received in operation 13304. In operation 13306, a view to launch is determined in response to the receipt of the notification and the reference. The view is based on predetermined criteria. The view is associated with the activity and displayed in operations 13308 and 13310.

Detailed Description Text (2164):
Because the View Configurer has pre-registered an interest in the startup of activities, it will receive a broadcast message. In this step, the View Configurer should receive a minimum of two parameters: Notification of the startup event that has just occurred. A reference to the new instance of the object that was just created.

Detailed Description Text (2166):
Benefits Development. Depending on the distribution model in place, business processing can be executed and tested before the appropriate views have been implemented. Automated testing. The View Configurer is particularly useful when you want to use scripts and avoid bringing up windows with automated testing. This is especially true for performance testing, where you might want to run 100 transactions, which might involve instantiating 100 instances of the same activity. Running processes in batch mode. The View Configurer allows processes to run without a View, and makes it very simple to connect, disconnect, or reconnect related views. Distribution Transparency. In a distributed environment, the process might live on a different machine from the end user's machine. In that case, it cannot launch the view directly, within its own executable. (Unless using a remote windowing system like X-windows, etc.) So the View Configurer allows application architects to transparently move process logic around, depending on the distribution model.

Detailed Description Text (2179):
Each assertion may be raised with descriptions for helping to identify where the assertion failed. Also, each assertion may be raised with parameters for helping to identify why the assertion failed. In one embodiment, two types of assertion classes may be provided. In such an embodiment, one of the assertion classes may implement assertion-checking logic and the other assertion class may implement only null operations, with one of the assertion classes being selected to be raised.

Detailed Description Text (2181):
Methods typically obtain and return a value, set an attribute based on a passed in parameter, or modify the state of the application based on some complex business rule or ruleset. While there is always some expected result of the invocation of an operation, there are also other, less expected possibilities. The provided parameters may not be within the expected range, thereby causing an error. A communications failure could cause the operation to fail to complete or, worse yet, return incorrect data or leave the system in an inconsistent state.

Detailed Description Text (2192):
Assertions can be raised with descriptions and parameters. A description can help to identify where the Assertion failed and a parameter list can help to identify why the Assertion failed.

Detailed Description Text (2197):
Benefits Ease ofError Identification. Many error are caused by invoking an operation with improper data (parameters). By formalizing these conditions, it is very obvious is an error was caused by bad data or bad code. Correctness. Properly placed assertions assure that the system is in a correct state and responses can be trusted. Assertion checking complements, but does not replace, a comprehensive testing program. The responsibility remains with the designer to identify the correct conditions to assert. Consistency. All checks will be made and handled in a similar fashion. Control. The enabling and disabling features of the Assertion allows an operations controller to determine when and what checks should be made at runtime rather then development time. Flexibility. All handling and clean-up of incorrect assertions is located in one place making changes to this logic much easier to implement. Readability. Polices concerning how assertions are actually thrown and handled is not in the functional code. Documentation. The code actually documents the design assumptions. This can also be used by documentation generators which read through the code.

Detailed Description Text (2204):
FIG. 138 illustrates a flowchart for a method 13800 for detecting an orphaned server context. A collection of outstanding server objects is maintained and a list of contexts is created for each of the outstanding server objects in operations 13802 and 13804. A compilation of clients who are interested in each of the outstanding server objects are added to the list in operation 13806. Recorded on the list in operation 13808 is a duration of time since the clients invoked a method accessing each of the contexts of the outstanding server objects. The list is examined at predetermined intervals for determining whether a predetermined amount of time has passed since each of the objects has been accessed in operation 13810. Contexts that have not been accessed in the predetermined amount of time are selected in operation 13812 and information is sent to the clients identifying the contexts that have not been accessed in the predetermined amount of time in operation 13814.

Detailed Description Text (2263):
Benefits Requirements Traceability. Exceptions requirements are captured and managed through implementation. Hierarchy Design. Analysis may show optimizations that can be made such as handling a subtree of exceptions with the same code, as the response is the same to any exception in the subtree. Interface Design. Discovery of interface requirements on the exception classes to support a particular response is another benefit. Handler design. Assists in exception handling design by identifying common responses that can be leveraged by the handlers.

Detailed Description Text (2335):
A key objective of a comprehensive object-to-relational persistence architecture is shielding the application business logic and developers from the relational structure. The benefits are a simplified environment for business developers, reduced distraction with technical issues, and increased focus on the business object model and functional logic. However, in order to reap these benefits, a significant investment in architecture development is typically required.

Detailed Description Text (2336):
The scope of a persistence architecture can range across the following levels of transparency and automation: Heavyweight, fully-automated, including the mapping of the object model to the database schema and generation of all the database access code. Variants of this architecture type may allow the customization of database access code (e.g., for optimization purposes). Lightweight mechanism which provides generic persistence capabilities to business objects but delegates all database access to separately developed data access routines. In this case, the data access routines are not part of the persistence architecture per se. Minimal persistence approach in which each business object is directly responsible for database access.

Detailed Description Text (2412):
Benefits Reuse. New transactions can take advantage of existing data access routines. For example, introducing a new business transaction, like perform credit check, would use existing customer and account objects. Yet, these domain model objects would already know how to update themselves. Therefore, the new application would build no

new data access code. Maintainability and extensibility. This approach supports "fix in one place." Any changes to particular data elements only need to be changed, tested, and recompiled in one access module, that of the owning business object. Uniformity. Both optimistic locking and referential integrity (RI) are typically defined at the business object level. For example, separate account and customer objects typically have their own locking attributes. In addition, an RI rule usually relates one entity to another, such as "all accounts must have a customer." Organizing data access around business entities simplifies locking and integrity. Both can use a uniform mechanism, implying that the architecture can hide technical details. This avoids the hard-coding typical of the transaction-based approach. Flexibility. Whole object retrieval is extensible. It allows a transaction to ask an object for any data. This supports maintenance and extension. A developer can easily change the particular data items a transaction uses. But whole retrieval still guarantees that those items will already have been retrieved. For example, a future release of the accept bill payment window could also display the social security number. Yet the data access routines would need no modification.

Detailed Description Text (2446):
Therefore, implement the associations using object identifiers that contain the necessary information to retrieve the object if it is needed. These objects can then be loaded when the object is restored, eliminating the need to restore the entire associated object. In addition, since these object identifiers uniquely identify an object instance, they can be used/passed in place of memory pointers. When the object is needed, simply restore the instance using the object identifier.

Detailed Description Text (2448):
The object identifier (or OID) must contain enough information to uniquely identify the instance. This identifier could be a unique row id generated by a database, a UUID generated by a utility or a unique string generated from one or more attributes. It is generally desirable to have a different class of OID for each type of object, thereby creating a more type-safe environment. It should also be noted that OID's should have value semantics.

Detailed Description Text (2467):
A dictionary can be used to implement the Object Identity Cache. The following points should be considering when implementing an Object Identify Cache.

Detailed Description Text (2485):
Given the use of a relational database as the persistent store, the scope of a persistence architecture can range across the following levels of transparency and automation: Heavyweight, fully-automated persistence architecture. Including the mapping of the object model to the database schema and generation of all the database access code Variants of the above scheme allowing the customization of database access code Lightweight mechanism which provides generic persistence capabilities to business objects but delegates all database access to separately developed data access routines. In this case, the data access routines are not part of the persistence architecture per se. Minimal persistence approach in which each business object is directly responsible for database access

Detailed Description Text (2487):
Data access and business logic are significantly different tasks both in their goals and development approaches. Consequently, except when a fully automated persistence architecture is used, it is often the case that two separate teams are responsible for the development of business logic and data access. If both teams work directly with the business objects, serious contention may result. Problems encountered in practice include: Changes to business logic that impacts the development (e.g., requires recompilation) of database access code even when there is no change to the attributes of business objects. (Note: recompilation can be a problem even if a fully automated persistence framework is used.) Changes to the database schema can impact the development of business objects The data access team m ay unduly influence the design of the business objects, leading to a data-centric model and design The two teams' development schedules need to be in sync; slippage on one team can adversely impact the other team's progress

Detailed Description Text (2507):
Object and component based projects designed and built from the ground up will likely have a well thought out component model and architecture where GUI widgets are linked or bound to domain objects. Data access (and retrieval) for these objects is organized around the business entity, rather than a transaction, and so data is packaged into

cross-functional objects, rather than transaction-specific data structures. Each business object manages the retrieval of its data items.

Detailed Description Text (2527):
A transaction is a set of business events that, coupled together, accomplish a particular business function, such as turning on gas service. Because the events are logically related, their data changes are logically related as well. Taken together, these data changes create a new, consistent state for the business model.

Detailed Description Text (2528):
While a transaction is in process, the state of the business model may not be consistent, so it is necessary to manage the entire transaction from its point of origin to its point of completion. Whether the transaction is successful or not, the point of completion will always result in a steady, consistent state for the business model. For successful transactions, data changes will be committed and the business model will reflect all new business data associated with the transaction. For failed transactions, data changes will be rolled back and the business model will appear as it did prior to the start of the transaction.

Detailed Description Text (2529):
To help manage the transaction from point of origin to point of completion, each transaction is organized through a single Logical Unit of Work (LUW). This LUW manages the business model and any of its subsequent data changes. While both users and internal exceptions can determine the success of a transaction, the LUW handles the commit and rollback operations.

Detailed Description Text (2531):
As transactions become more complex and require a greater scale of changes to the business model, the LUW trying to manage these changes becomes large and unwieldy. To simplify these transactions, the LUW is broken down into nested, more granular, logically related units of work called Secondary LUWs. Secondary LUWs are identical to LUWs except that their commit and rollback operations affect only the business model of their parent LUW and are not persisted to a data store. Consequently, a secondary LUW must manage its data changes differently than its parent.

Detailed Description Text (2533):
One method for managing changes to the business model involves copying the model into the secondary LUW. Another often simpler approach is to store both old and new (or potential) values for all objects in the business model.

Detailed Description Text (2535):
In the process of managing its business model, a LUW will often have to send messages to all business objects within the LUW. Examples of such messages include saveDataChanges, retrieve, or isDirty. Rather than hardcoding a call to each object in the model, the pattern LUW Context suggests using a bag (or collection) to hold all objects referenced by the LUW. Then, a single message can be sent to the bag which will forward it to all objects within it.

Detailed Description Text (2536):
Support for user multi-tasking can also present problems for LUWs. Through multi-tasking, multiple LUWs will be running concurrently. Problems occur if the business models of these concurrent LUWs overlap and the transactions attempt to write to the same business object. A call center representative trying to solve two customer problems during the course of one call is one example of this scenario. The Separate Models pattern helps solve this issue by assigning each LUW independent copies of their portion of the business model. This keeps one transaction from interfering with another.

Detailed Description Text (2538):
When an LUW is called to commit, the transaction will assemble the necessary objects from the business model to send their data changes to the information services layer. This group of objects will include all new and dirty objects as well as any objects marked for deletion. For each business object, the transaction will likely have a corresponding request. If each of these requests were then sent to information services independently, a large number of network messages would result. To solve this problem, the Request Batcher pattern batches all requests associated with a transaction together into one network message. On the other end of the network, Information services would unbatch the transaction requests and persist the data changes.

Detailed Description Text (2558):
Benefits Performance. This solution supports request batching for retrieving interdependent models. Reuse. Because dependencies are not hard-coded, business objects can be reused independently of each other. Loose Coupling. When a request dynamically registers its dependency, it need not know anything about its parent request. The dependent effectively says, "I don't know who you are, but I know that your response data contains my identifier."

Detailed Description Text (2564):
Optionally, the receiver may hide technical details including details of persistence and garbage collection from business developers. As a further option, the business objects may be distributed across a network. Also, the receiver may distribute the message to each of the business objects across the network. Additionally, the logical unit of work may optionally be modeled as an object in software.

Detailed Description Text (2569):
A retrieveData( ) message might also be required, if the object model pre-instantiates objects before retrieving them. Similarly, refresh( ) could be used: the business object, if dirty, replaces any changes with data originally from the data store.

Detailed Description Text (2579):
Thus, the scope of a bag is an LUW. In addition, a bag provides contextual information for the LUW--i.e., which business objects that LUW uses. The architecture bag therefore models the LUW Context, and will be named as such.

Detailed Description Text (2581):
For example, consider a search window which has instantiated 30 business objects. Releasing those objects, if the messages were sent independently, would require 30 network messages. However, with LUW Context, a single message can go from the client to the server. Then, within the server executable, the LUW Context forwards release( ) to the 30 member objects. This is far less costly than using the network for that messaging. Because of this message batching, some readers may confuse LUW Context with Request Batcher. It is true that both reduce the number of network messages. However, the former is concerned with supporting a family of generic, architecture messages, like isDirty( ) and refresh( ), on a single atomic object. The latter is concerned with grouping database requests into a physical package, for un-batching at the server. Although both have similar principles and characteristics, they solve different problems and are implemented differently. Architecture Extensibility. LUW Context models the LUW as an actual object in the software. Any other architecture processing which executes on a per-LUW basis can also be coded there. (See the Related Patterns section for examples.)

Detailed Description Text (2584):
An LUW Context can collaborate with a Request Batcher, if requests are batched for transmission to the data store. Rather than storing the batcher globally, each context and hence model can have its own manager. This allows multiple domain models, in multiple contexts, to send transactions simultaneously but independently. Then, whenever a business object requests an access or update, its request will be intercepted by the model's particular Request Batcher. The batcher then holds these requests until the activity--which owns the LUW--tells the batcher to send them.

Detailed Description Text (2602):
FIG. 188 illustrates a flowchart for a method 18800 for sorting requests that are being unbatched from a batched message. A group of business objects necessary for a transaction are provided in operation 18802. Logically-related requests received from the business objects are grouped in operation 18804. Sorting rules and/or sort weights are obtained in operation 18806 and, in operation 18808, the requests in the message are sorted and placed in a specific order determined from the sorting rules and/or the sort weights. The sorted requests are batched into a single message which is sent to a data server where the requests are unbundled from the message in the specific order (see operations 18810, 18812, and 18814).

Detailed Description Text (2603):
A request may also not be allowed to proceed until all dependent requests are completed. A plurality of transactions may each use the same sorting rules for preventing deadlocks. Optionally, the class represented by each request may be determined so that the sorting rules may be based on a class type. As another option, the sorting rules may include referential integrity rules which ensure that references between two relational tables are valid. In such a situation, a linear ordering of

requests may also be created based on the referential integrity rules. The numbering of the position of the request in the linear ordering may also be the weight of that request so that requests with lower weights are processed before requests with higher weights.

Detailed Description Text (2606):
Referential Integrity (RI) ensures that references between two relational tables are valid. That is, foreign keys in one table must refer to existing primary keys in another table. For example, RI rules could require that all accounts have a customer. Then, values in account.cust_id would need matching values in customer.cust_id.

Detailed Description Text (2614):
Traditionally, transactions have hard-coded deadlock avoidance and RI. Each transaction has called its own update module. Each hand-crafted module has ordered multiple SQL statements, according to these rules.

Detailed Description Text (2617):
Therefore, an update transaction should sort its requests before sending them to the data server. The sorted result will conform to RI rules. Then, across update transactions, all customer requests can appear before all account requests. In addition, every transaction will use the same sort algorithm. That will prevent deadlocks.

Detailed Description Text (2620):
The sorter 19000 will have visibility to sorting rules, or even weights, to determine this order. The rules can typically be based on the class type. Before sending the transaction, the sorter can ask each request which class it represents. In this manner, the sorter can re-order the requests appropriately.

Detailed Description Text (2621):
Benefits Separation of Concern. This sorting pattern hides the technical details and complexity of RI from business logic. Applications avoid hard-wiring customized RI rules for its transactions. Maintainability. RI rules can easily be changed without impacting application code. Granted, this does not happen frequently in production. Reusability. The generic Request Sorter uses universal sorting rules, or weights. These rules are global across business processes. Moreover, the rules are based on existing, reusable business objects. Therefore, new applications can reuse the sorter, as well. Visibility. If RI enforcement is distributed across application logic, it can be difficult to get a complete picture of the referential rules. Request Sorter centralises those rules (i.e. weights) in one, visible place.

Detailed Description Text (2622):
A complete, linear ordering of all domain classes can be created, based on the RI rules. Each class will have a unique position in the ordering. This position is the class' weight for the sorting algorithm. Requests for domain objects with lower weights will always appear before requests with higher weights.

Detailed Description Text (2624):
This satisfies the RI rule mentioned earlier, because Customers have a lower weight (28) than Accounts (30). Thus, requests for customers will appear in any transaction before requests for accounts. As long as the order satisfies every RI rule, the request sorter can use such a linear ordering.

Detailed Description Text (2634):
However, an MVC-based OO architecture does not naturally support this requirement. With MVC, the domain model stores all data changes. Windows are merely a view into this model, and they have little business data of their own. In addition, MVC model objects have no idea which views are using them. Instead, the model anonymously broadcasts its data changes, and all views on the model respond by updating themselves. This synchronizes windows with their business data. Thus, MVC allows multiple views to simultaneously display, and be refreshed by, a single copy of the model data.

Detailed Description Text (2636):
Unfortunately, this benefit of MVC introduces a problem. A globally-shared domain model does not naturally separate concurrent LUWs. It puts a burden on business "activity" objects, which coordinate the high-level business processing across their domain models. Each activity has to either avoid overlap or know specifically how it affects the model.

Detailed Description Text (2645):
Rather than using a globally-shared model, each business LUW will own a private, scratchpad copy of its domain model. This satisfies the independence requirement. A business object in one model will automatically be a separate instance from a business object in another model, even if they share the same functional identity. For example, simultaneously opened payment and services windows would have separate copies of Account 101.

Detailed Description Text (2647):
FIG. 193 illustrates the Separate Models for Separate Business LUWs 19300,19302.

Detailed Description Text (2650):
Thus, using Separate Models preserves the integrity of business LUWs. It allows each LUW to easily save or cancel independently.

Detailed Description Text (2653):
Benefits Isolation. Most fundamentally, this pattern solves the Isolation requirement of ACID. It ensures that each LUW has its own "working storage" copies of business data. Transparency. Separating models can be done in an architected fashion, as outlined in the implementation section. The separation of LUWs--which is a technical issue--can be hidden completely from business logic. Imagine instead that LUWs didn't have their own copies. Then, each operation might need an additional argument: the LUW owning the data change. This would pollute application code with an extra "transaction ID" argument, as in setBalance(newBalance, transactionId). As previously mentioned, this is only required in the absence of an Object Transaction Monitor. An OTM can transparently manage the transaction Id with the thread, without including it as an explicit argument. Uniformity. Application developers don't need to know about which objects may or may not be used by other, concurrent LUWs.

Detailed Description Text (2656):
In addition, each business object can point to its context. In that manner, business objects know their LUW. This could be useful, for example, while building a domain model. Then, the parent object could propagate its context to a linked, child object.

Detailed Description Text (2657):
Business objects owned by the same business LUW share the same LUW Context, whereas different LUWs have different contexts. Each context therefore contains its own "working-storage" copy of the model. This delineates an individual workspace, or scratchpad, for each LUW.

Detailed Description Paragraph Table (26):
```
public Vector validate() { Vector errors = new Vector(); Hashtable rulesAndWidgets =
this.getRulesAndWidgets(); Enumeration rules = rulesAndWidgets.keys(); while
(rules.hasMoreElements()) { ValidationRule aRule = (ValidationRule)rules.nextElement();
ValidateWidget aWidget = (ValidateWidget) RulesAndWidgets.get(aRule); String anError =
aWidget.validateRule(aRule); if (anError != null) { errors.addElement(anError); } }
return errors; }
```

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Terms	Documents
L9 and rul$3	54

Database: US Patents Full-Text Database; US Pre-Grant Publication Full-Text Database; JPO Abstracts Database; EPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Search: L11

Refine Search

Recall Text | Clear

Search History

DATE: Monday, December 30, 2002 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ		
L11	L9 and rul$3	54	L11
L10	L9 and (match$3 or mirror$3) same (categor$6 or optimiz$3)	5	L10
L9	L8 and identify$	81	L9
L8	L7 and (receiv$3 or generat$3) same (event or occur$6)	82	L8
L7	L6 and execut$3 same interact$3	105	L7
L6	L1 and model$6 same (business or auction$6 or transact$6)	474	L6
L5	L4 and rul$3 same engine	2	L5
L4	L3 and uniform same defin$6	3	L4
L3	L1 and plurality same model$6 same (business or auction$6 or transact$6)	41	L3
L2	plurality same model$6 same (business or auction$6 or transact$6) same domain	9	L2
L1	(ontolog$6 or collect$6 or concept$6) same driv$3 same (info or information or data or fil$3)	28485	L1

END OF SEARCH HISTORY

L4: Entry 12 of 27 File: PGPB Mar 28, 2002

DOCUMENT-IDENTIFIER: US 20020038217 A1
TITLE: System and method for integrated data analysis and management

Detail Description Paragraph (3):
[0014] The disclosed methods of analyzing and managing business data preferably operate in a system having an architecture such as that illustrated in FIG. 1. As shown, the system 100 includes a workflow manager 110 that is responsible for collecting data from several disparate data sources, analyzing the collected information and presenting the results to a user via a contextual visualization interface 112. Workflow manager 110 may be employed to provide a number of business solutions related to, for example, supply chain management, manufacturing optimization, distribution and warehousing. The relevant business data is collected from other sources in a number of ways, including for example via a global computer network 112, such as the Internet, via a wide area network (WAN) or local area networks (LAN) connection, via batch processing and via a human operator.

Detail Description Paragraph (8):
[0019] Contextual visualization interface 112 enables a user to view various aspects of the a business. The visualization management service provides a lower level management capability aimed at audiences where a bigger management and status picture is required. In a preferred embodiment, it utilizes the Unicenter TNG WorldView interface developed by Computer Associates International, Inc. with a business object model that implements Business Process Views (BPV's) representing the different aspects of an enterprise such as manufacturing, finance and human resources, for example. Preferably, the business object model is an extensible object model that implements both line-of-business visualization object definitions and application-level data object definitions.

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Terms	Documents
L1 and ontolog$6	2

Database: US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L6

Refine Search

Recall Text | Clear

Search History

DATE: Monday, December 30, 2002 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT,PGPB,JPAB,EPAB,DWPI,TDBD; PLUR=YES; OP=ADJ			
L6	L1 and ontolog$6	2	L6
L5	L4 and business same rul$3	16	L5
L4	l1 and (ontolog$6 or context$6 or concept$6) same model$3	27	L4
L3	L2 and (chang$6 or transform$3) same (intelligence or ic or smart card)	16	L3
L2	l1 and (ontolog$6 or context$6 or concept$6) same model$3 same (business$3 or transact$6)	17	L2
L1	(www or world wide web or e-commerce or electronic commerce or internet) same analyz$3 same distribut$3 same (data or fil$3 or info or information)	213	L1

END OF SEARCH HISTORY